

# Automatic Completion of Distributed Protocols with Symmetry

Rajeev Alur<sup>†</sup>, Mukund Raghothaman<sup>†</sup>, Christos Stergiou<sup>‡</sup>,  
Stavros Tripakis<sup>‡</sup>, and Abhishek Udupa<sup>†</sup>

<sup>†</sup>University of Pennsylvania, Philadelphia, USA

<sup>‡</sup>University of California, Berkeley, USA

<sup>#</sup>Aalto University, Helsinki, Finland

## Abstract

A distributed protocol is typically modeled as a set of communicating processes, where each process is described as an extended state machine along with fairness assumptions, and its correctness is specified using safety and liveness requirements. Designing correct distributed protocols is a challenging task. Aimed at simplifying this task, we allow the designer to leave some of the guards and updates to state variables in the description of extended state machines as unknown functions. The protocol completion problem then is to find interpretations for these unknown functions while guaranteeing correctness. In many distributed protocols, process behaviors are naturally symmetric, and thus, synthesized expressions are further required to obey symmetry constraints. Our counterexample-guided synthesis algorithm consists of repeatedly invoking two phases. In the first phase, candidates for unknown expressions are generated using the SMT solver Z3. This phase requires carefully orchestrating constraints to enforce the desired symmetry in read/write accesses. In the second phase, the resulting completed protocol is checked for correctness using a custom-built model checker that handles fairness assumptions, safety and liveness requirements, and exploits symmetry. When model checking fails, our tool examines a set of counterexamples to safety/liveness properties to generate constraints on unknown functions that must be satisfied by subsequent completions. For evaluation, we show that our prototype is able to automatically discover interesting missing details in distributed protocols for mutual exclusion, self stabilization, and cache coherence.

## 1 Introduction

Protocols for coordination among concurrent processes are an essential component of modern multiprocessor and distributed systems. The multitude of behaviors arising due to asynchrony and concurrency makes the design of such protocols difficult. Consequently, analyzing such protocols has been a central theme of research in formal verification for decades. Now that verification tools are mature enough to be applied to find bugs in real-world protocols, a promising area of research is *protocol synthesis*, aimed at simplifying the design process via more intuitive programming abstractions to specify the desired behavior.

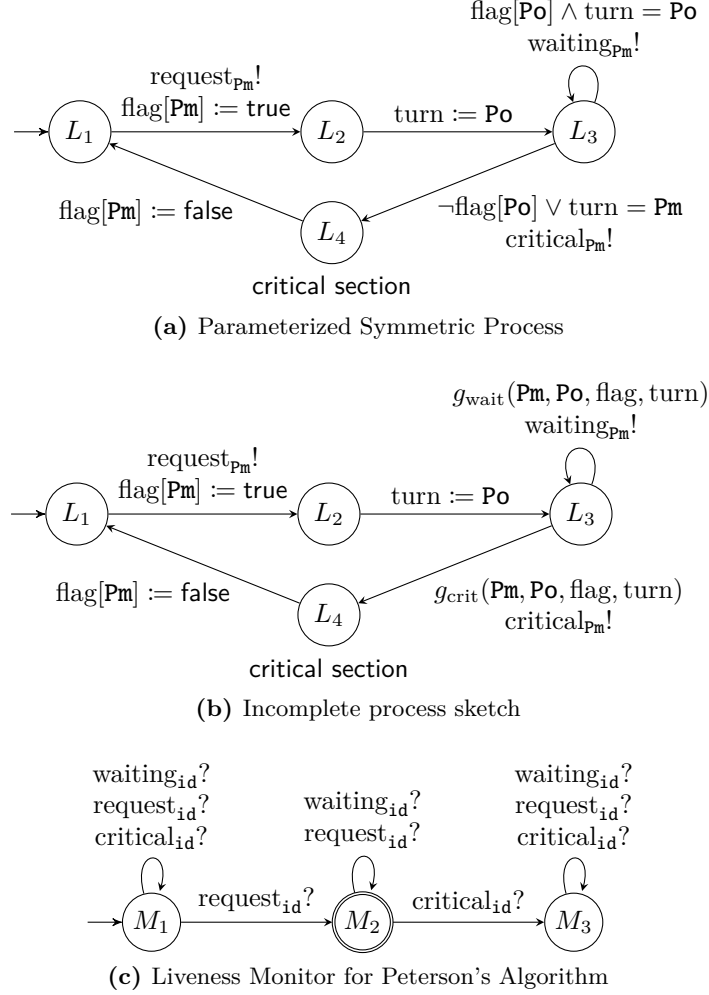
Traditionally, a distributed protocol is modeled as a set of communicating processes, where each process is described by an extended state machine. The correctness is specified by both safety and liveness requirements. In *reactive synthesis* [4, 22, 24], the goal is to automatically derive a protocol from its correctness requirements specified in temporal logic. However, if we require the implementation to be distributed, then reactive synthesis is undecidable [12, 19, 23, 30]. An alternative, and potentially more feasible approach inspired by

*program sketching* [27], is to ask the programmer to specify the protocol as a set of communicating state machines, but allow some of the guards and updates to state variables to be unknown functions, to be completed by the synthesizer so as to satisfy all the correctness requirements. This methodology for protocol specification can be viewed as a fruitful collaboration between the designer and the synthesis tool: the programmer has to describe the structure of the desired protocol, but some details that the programmer is unsure about, for instance, regarding corner cases and handling of unexpected messages, are filled in automatically by the tool.

In our formalization of the synthesis problem, processes communicate using input/output channels that carry typed messages. Each process is described by a state machine with a set of typed state variables. Transitions consist of guards that test input messages and state variables and updates to state variables and fields of messages to be sent. Such guards and updates can involve *unknown* (typed) functions to be filled in by the synthesizer. In many distributed protocols, such as cache coherence protocols, processes are expected to behave in a symmetric manner. Thus, we allow variables to have *symmetric types* that restrict the read/write accesses to obey symmetry constraints. To specify safety and liveness requirements, the state machines can be augmented with acceptance conditions that capture incorrect executions. Finally, fairness assumptions are added to restrict incorrect executions to those that are *fair*. It is worth noting that in verification one can get useful analysis results by focusing solely on safety requirements. In synthesis, however, ignoring liveness requirements and fairness assumptions, typically results in trivial solutions. The protocol completion problem, then, is, given a set of extended state machines with unknown guards and update functions, to find expressions for the unknown functions so that the composition of the resulting machines does not have an accepting fair execution.

Our synthesis algorithm relies on a counterexample-guided strategy with two interacting phases: candidate interpretations for unknown functions are generated using the SMT solver Z3 and the resulting completed protocol is verified using a model checker. We believe that our realization of this strategy leads to the following contributions. First, while searching for candidate interpretations for unknown functions, we need to generate constraints that enforce symmetry in an accurate manner without choking current SMT solvers. Second, surprisingly there is no publicly available model checker that handles all the features that we critically need, namely, symmetry, liveness requirements, and fairness assumptions. As a result, building on the known theoretical foundations, we had to develop our own model checker (which we plan to make publicly available). Third, we develop an algorithm that examines the counterexamples to safety/liveness requirements when model checking fails, and generates constraints on unknown functions that must be satisfied in subsequent completions. Finally, the huge search space for candidate expressions is a challenge for the scalability for any synthesis approach. As reported in section 4, we experimented with many alternative strategies for prioritizing the search for candidate expressions, and this experience offers some insights regarding what information a user can provide for getting useful results from the synthesis tool. We evaluate our synthesis tool in completing a mutual exclusion protocol, a self stabilization protocol and a non-trivial cache coherence protocol. Large parts of the behavior of the protocol were left unspecified in the case of the mutual exclusion protocol and the self stabilization protocol, whereas the cache coherence protocol had quite a few tricky details left unspecified. Our tool was able to synthesize the correct completions for all these protocols in a reasonable amount of time.

**Related Work.** *Bounded synthesis* [13] and *genetic programming* [17, 18] are other approaches for handling the undecidability of distributed reactive synthesis. In the first, the size of the implementation is restricted, thus allowing for algorithmic solutions, and in the second the implementation space is sampled and candidates are mutated in a stochastic process. The problem of inferring extended finite-state machines has been studied in the context of active learning [5]. The problem of completing distributed protocols has been targeted by the works presented in [2, 31] and *program repair* [16] addresses a similar problem. Compared to [2], our algorithm can handle extended state machines that include variables and transitions with symbolic expressions as guards and updates. Compared to [31], our algorithm can also handle liveness violations and,



**Figure 1:** Peterson's mutual exclusion algorithm. Note that the non-trivial guards of the  $(L_3, L_3)$  and  $(L_3, L_4)$  transitions in Figure 1(a) have been replaced in Figure 1(b) by “unknown” functions  $g_{\text{wait}}$  and  $g_{\text{crit}}$  respectively.

more importantly, can process counterexamples automatically and, thus, does not require a human in the synthesis loop. PSKETCH [28] is an extension of the *program sketching* work for concurrent data structures but is limited to safety properties. The work in [14] describes an approach based on QBF solvers for synthesizing a distributed self-stabilizing system, which also approximates liveness with safety and uses templates for the synthesized functions. Last, compared to all works mentioned above, our algorithm can be used to enforce symmetry in the synthesized processes.

## 2 An Illustrative Example

Consider Peterson's mutual exclusion algorithm, described in Figure 1(a), which manages two symmetric processes contending for access to a critical section. Each process is parameterized by  $\mathbf{Pm}$  and  $\mathbf{Po}$  (for “my” process id and “other” process id respectively), such that  $\mathbf{Pm} \neq \mathbf{Po}$ . Both parameters  $\mathbf{Pm}$  and  $\mathbf{Po}$  are of type *processid* and they are allowed to take on values  $\mathbf{P0}$  and  $\mathbf{P1}$ . We therefore have two instances:  $P_0$ , where

$(P_m = P_0, P_o = P_1)$ , and  $P_1$ , where  $(P_m = P_1, P_o = P_0)$ .  $P_0$  and  $P_1$  communicate through the shared variables *turn* and *flag*. The variable *turn* has type `processid`. The *flag* variable is an array of Boolean values, with index type `processid`. The main objective of the protocol is to control access to the critical section, represented by location  $L_4$ , and ensure that both of the processes  $P_0$  and  $P_1$  are never simultaneously in the critical section, *i.e.*, it is a safety violation for both  $P_0$  and  $P_1$  to be in state  $L_4$  at the same time.

The liveness monitor shown in Figure 1(c) captures the requirement that a process does not wait indefinitely to enter the critical section. The monitor accepts all undesirable runs where a process has requested access to the critical section but never reaches state  $L_4$  after. The messages request, waiting, and critical inform the liveness monitor about the state of the processes, and the synchronization model here is that of communicating I/O automata [20]. Note that a run accepted by the monitor may be *unfair* with respect to some processes. Enforcing *weak* process fairness on  $P_0$  and  $P_1$ , — *i.e.*, if a process is enabled at every point in an accepting cycle, then it must be executed at some point in the cycle — is sufficient to rule out unfair executions, but not necessary. Enforcing weak fairness on the transitions between  $(L_2, L_3)$ ,  $(L_3, L_4)$  and  $(L_4, L_1)$  suffices.

Now, suppose the protocol developer has trouble figuring out the exact guard under which a process is allowed to enter the critical section, but knows the structure of the processes  $P_0$  and  $P_1$ , and requires them to be symmetric. Figure 1(b) describes what the developer knows about the protocol. The functions  $g_{\text{wait}}$  and  $g_{\text{crit}}$  represent unknown Boolean valued functions over the state variables and the parameters of the process under consideration. Including the parameters as part of the domain of  $g_{\text{wait}}$  and  $g_{\text{crit}}$  indicates that the completions for processes  $P_0$  and  $P_1$  need to be symmetric. The objective is to assist the developer by automatically discovering interpretations for these unknown functions, such that the completed protocol satisfies the necessary mutual exclusion property, and the requirements imposed by the liveness monitor. We formalize this completion problem in Section 3, and present our completion algorithm in Section 4.

## 3 Formalization

### 3.1 Extended State Machine Sketches

We model processes using Extended State Machine Sketches (ESM-S). Fix a collection of types, such as the type *bool* of the Boolean values  $\{\text{true}, \text{false}\}$ , enumerated types such as  $\{\text{red}, \text{green}, \text{blue}\}$ , or finite subsets  $\text{nat}[x, y]$  of natural numbers  $\{i \mid x \leq i \leq y\}$ . Other examples of types include *symmetric types* (described in Section 3.2), and array types, which map each value of the index type to a value of the range type. Note that the cardinality of each type is required to be finite.

The description of an ESM-S will mention several function symbols. Some of these, such as the operator “+” for addition, have interpretations which are already known, while others, such as the guard  $g_{\text{crit}}$  to enter the critical section in the incomplete sketch of Peterson’s algorithm have unknown interpretations. Each function symbol, both known and unknown, is associated with a signature,  $d_1 \times \dots \times d_n \rightarrow r$ , where  $d_1, \dots, d_n$  are the types of its arguments and  $r$  is the return type. Expressions may then be constructed, such as “*timestamp* + 1”, using these function symbols, state variables, and input channels. Formally, an ESM-S  $A$  is a tuple  $\langle L, l_0, I, O, S, \sigma_0, U, T, \mathcal{F}_s, \mathcal{F}_w \rangle$  such that:

- $L$  is a finite set of locations and  $l_0 \in L$  is the initial location,
- $I$  and  $O$  are finite sets of typed input and output channels, respectively,
- $S$  is a finite set of typed state variables,
- $\sigma_0$  maps each variable  $x \in S$  to its initial value  $\sigma_0(x)$ ,
- $U$  is a set of unknown function symbols,
- $T$  is a set of transitions of the form  $\langle l, c, \text{guard}, \text{update}, l' \rangle$ , where  $c \in I$  for input,  $c \in O$  for output, and  $c = \epsilon$  for internal transitions, **guard** is the transition guard, and **update** are the transition updates,

- $\mathcal{F}_s, \mathcal{F}_w \subseteq 2^{T_e \cup T_O}$ , are sets of strong and weak fairnesses respectively. Here  $T_O$  and  $T_e$  are the sets of output and internal transitions respectively.

A guard description **guard** is a Boolean expression over the state variables  $S$  that can use unknown functions from  $U$ . Similarly, an update description **update** is a sequence of assignments of the form  $lhs := rhs$  where  $lhs$  is one of the state variables or an output channel in the case of an output transition, and  $rhs$  is an expression over state variables or state variables and an input channel in the case of an input transition, possibly using unknown functions from  $U$ .

**Executions.** To define the executions of an ESM-S, we first pick an *interpretation*  $R$  which maps each unknown function  $u \in U$  to an interpretation of  $u$ . Given a set of variables  $V$ , a *valuation*  $\sigma$  is a function which maps each variable  $x \in V$  to a value  $\sigma(x)$  of the corresponding type, and we write  $\Sigma_V$  for the set of all such valuations. Given a valuation  $\sigma \in \Sigma_V$ , a variable  $x$ , and a value  $v$  of appropriate type, we write  $\sigma[x \mapsto v] \in \Sigma_{V \cup \{x\}}$  for the valuation which maps all variables  $y \neq x$  to  $\sigma(y)$ , and maps  $x$  to  $v$ .

The executions of  $A$  are defined by describing the updates to the state valuation  $\sigma \in \Sigma_S$  during each transition. Note that each guard description **guard** naturally defines a set  $\llbracket \text{guard}, R \rrbracket$  of valuations  $\sigma \in \Sigma_S$  which satisfy **guard** with the unknown functions instantiated with  $R$ . Similarly, each update description **update** defines a function  $\llbracket \text{update}, R \rrbracket$  of type  $\Sigma_{S \cup \{x\}} \rightarrow \Sigma_S$  for input transitions on the channel  $x$ ,  $\Sigma_S \rightarrow \Sigma_{S \cup \{y\}}$  for output transitions on the channel  $y$ , and  $\Sigma_S \rightarrow \Sigma_S$  for internal transitions respectively.

A *state* of an ESM-S  $A$  is a pair  $(l, \sigma)$  of a location  $l \in L$  and a state valuation  $\sigma \in \Sigma_S$ . We then write:

- $(l, \sigma) \xrightarrow{x?v} (l', \sigma')$  if  $A$  has an input transition from  $l$  to  $l'$  on channel  $x$  with guard **guard** and update **update** such that  $\sigma \in \llbracket \text{guard}, R \rrbracket$  and  $\llbracket \text{update}, R \rrbracket(\sigma[x \mapsto v]) = \sigma'$ ;
- $(l, \sigma) \xrightarrow{y!v} (l', \sigma')$  if  $A$  has an output transition from  $l$  to  $l'$  on channel  $y$  with guard **guard** and update **update** such that  $\sigma \in \llbracket \text{guard}, R \rrbracket$  and  $\llbracket \text{update}, R \rrbracket(\sigma) = \sigma'[y \mapsto v]$ ; and
- $(l, \sigma) \xrightarrow{\epsilon} (l', \sigma')$  if  $A$  has an internal transition from  $l$  to  $l'$  with guard **guard** and update **guard** such that  $\sigma \in \llbracket \text{guard}, R \rrbracket$  and  $\llbracket \text{update}, R \rrbracket(\sigma) = \sigma'$ .

We write  $(l, \sigma) \rightarrow (l', \sigma')$  if either there are  $x, v$  such that  $(l, \sigma) \xrightarrow{x?v} (l', \sigma')$ , there are  $y, v$  such that  $(l, \sigma) \xrightarrow{y!v} (l', \sigma')$ , or  $(l, \sigma) \xrightarrow{\epsilon} (l', \sigma')$ . A finite (infinite) *execution* of the ESM-S  $A$  under  $R$  is then a finite (resp. infinite) sequence:  $(l_0, \sigma_0) \rightarrow (l_1, \sigma_1) \rightarrow (l_2, \sigma_2) \rightarrow \dots$  where for every  $j \geq 0$ ,  $(l_j, \sigma_j)$  is a state of  $A$ ,  $(l_0, \sigma_0)$  is an initial state of  $A$ , and for  $j \geq 1$ ,  $(l_j, \sigma_j) \rightarrow (l_{j+1}, \sigma_{j+1})$ . A state  $(l, \sigma)$  is *reachable* under  $R$  if there exists a finite execution that reaches that state:  $(l_0, \sigma_0) \rightarrow \dots \rightarrow (l, \sigma)$ . We say that a transition from  $l$  to  $l'$  with guard **guard** is *enabled* in state  $(l, \sigma)$  if  $\sigma \in \llbracket \text{guard}, R \rrbracket$ . A state  $(l, \sigma)$  is called a *deadlock* if no transition is enabled in  $(l, \sigma)$ . The ESM-S  $A$  is called *deadlock-free* under  $R$  if no deadlock state is reachable under  $R$ . The ESM-S  $A$  is called *deterministic* under  $R$  if for every state  $(l, \sigma)$ , if there are multiple transitions enabled at  $(l, \sigma)$ , then they must be input transitions on distinct input channels.

Consider a weak fairness requirement  $F \in \mathcal{F}_w$ . An infinite execution of  $A$  under  $R$  is called *fair* with respect to a weak fairness  $F$  if either: (a) for infinitely many indices  $i$ , none of the transitions  $t \in F$  is enabled in  $(l_i, \sigma_i)$ , or (b) for infinitely many indices  $j$  one of the transitions in  $F$  is taken at step  $j$ . Thus, for example, the necessary fairness assumptions for Peterson's algorithm are  $\mathcal{F}_w = \{\{\tau_{23}\}, \{\tau_{34}\}, \{\tau_{41}\}\}$ , where  $\tau_{23}$ ,  $\tau_{34}$ , and  $\tau_{41}$  refer to the  $(L_2, L_3)$ ,  $(L_3, L_4)$  and  $(L_4, L_1)$  transitions respectively. Similarly, an infinite execution of  $A$  under  $R$  is fair with respect to a strong fairness  $F \in \mathcal{F}_s$  if either: (a) there exists  $k$  such that for every  $i \geq k$  and every transition  $t \in F$ ,  $t$  is not enabled in  $(l_i, \sigma_i)$ , or (b) for infinitely many indices  $j$  one of the transitions in  $F$  is taken at step  $j$ . Finally, an infinite execution of  $A$  is fair if it is fair with respect to each strong and weak fairness requirement in  $\mathcal{F}_s$  and  $\mathcal{F}_w$  respectively.

**Composition of ESM-S** Informally, two ESM-S  $A_1$  and  $A_2$  are composed by synchronizing their output and input transitions on a given channel. If  $A_1$  has an output transition on channel  $c$  from location  $l_1$  to  $l'_1$  with guard and updates **guard**<sub>1</sub> and **update**<sub>1</sub>, and  $A_2$  has an input transition on the same channel  $c$  from location  $l_2$  to  $l'_2$  with guard and updates **guard**<sub>2</sub> and **update**<sub>2</sub> then their product has an output transition

from location  $(l_1, l_2)$  to  $(l'_1, l'_2)$  on channel  $c$  with guard  $\text{guard}_1 \wedge \text{guard}_2$  and updates  $\text{update}_1; \text{update}_2$ . Note that by sequencing the updates, the value written to the channel  $c$  by  $A_1$  is then used by subsequent updates of the variables of  $A_2$  in  $\text{update}_2$ . We now provide a formal definition of the composition of two ESM sketches.

Consider two ESM-S  $A_1$  and  $A_2$ , where  $A_1$  is of the form  $A_1 = \langle L_1, l_{0,1}, I_1, O_1, S_1, \sigma_{0,1}, U_1, T_1, \mathcal{F}_{s1}, \mathcal{F}_{w1} \rangle$  and  $A_2$  is of the form  $A_2 = \langle L_2, l_{0,2}, I_2, O_2, S_2, \sigma_{0,2}, U_2, T_2, \mathcal{F}_{s2}, \mathcal{F}_{w2} \rangle$  such that  $O_1 \cap O_2 = \emptyset$  and  $S_1 \cap S_2 = \emptyset$ . We then define their composition,  $A_1 \mid A_2$ , as the ESM-S:  $\langle L, l_0, I, O, S, \sigma_0, U, T, \mathcal{F}_s, \mathcal{F}_w \rangle$  where:

- $L = L_1 \times L_2$ ,
- $l_0 = \langle l_{0,1}, l_{0,2} \rangle$ ,
- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ ,
- $O = O_1 \cup O_2$ ,
- $S = S_1 \cup S_2$ ,
- $\sigma_0 = \sigma_{0,1} \cup \sigma_{0,2}$ ,
- $U = U_1 \cup U_2$ ,
- For every input channel  $x \in I$ ,  $\langle (l_1, l_2), x, \text{guard}, \text{update}, (l'_1, l'_2) \rangle \in T$  if and only if *at least* one of the following holds:
  - (a)  $x \notin I_2$ ,  $l_2 = l'_2$ , and  $\langle l_1, x, \text{guard}, \text{update}, l'_1 \rangle \in T_1$ ,
  - (b)  $x \notin I_1$ ,  $l_1 = l'_1$ , and  $\langle l_2, x, \text{guard}, \text{update}, l'_2 \rangle \in T_2$ ,
  - (c)  $x \in I_1 \cap I_2$ ,  $\langle l_1, x, \text{guard}_1, \text{update}_1, l'_1 \rangle \in T_1$ ,  $\langle l_2, \text{guard}_1, \text{update}_1, l'_2 \rangle \in T_2$ ,  $\text{guard} = \text{guard}_1 \wedge \text{guard}_2$ , and  $\text{update} = \text{update}_1; \text{update}_2$ ,
- For every output channel  $y \in O$ ,  $\langle (l_1, l_2), y, \text{guard}, \text{update}, (l'_1, l'_2) \rangle \in T$  if and only if *at least* one of the following holds:
  - (a)  $y \in O_1$ ,  $y \notin I_2$ ,  $l_2 = l'_2$ , and  $\langle l_1, y, \text{guard}, \text{update}, l'_1 \rangle \in T_1$ ,
  - (b)  $y \in O_2$ ,  $y \notin I_1$ ,  $l_1 = l'_1$ , and  $\langle l_2, y, \text{guard}, \text{update}, l'_2 \rangle \in T_2$ ,
  - (c)  $y \in O_1$ ,  $y \in I_2$ ,  $\langle l_1, \text{guard}_1, \text{update}_1, l'_1 \rangle \in T_1$ ,  $\langle l_2, \text{guard}_2, \text{update}_2, l'_2 \rangle \in T_2$ ,  $\text{guard} = \text{guard}_1 \wedge \text{guard}_2$ , and  $\text{update} = \text{update}_1; \text{update}_2$ ,
  - (d)  $y \in O_2$ ,  $y \in I_1$ ,  $\langle l_1, y, \text{guard}_1, \text{update}_1, l'_1 \rangle \in T_1$ ,  $\langle l_2, y, \text{guard}_2, \text{update}_2, l'_2 \rangle \in T_2$ ,  $\text{guard} = \text{guard}_1 \wedge \text{guard}_2$ , and  $\text{update} = \text{update}_2; \text{update}_1$ ,
- $\langle (l_1, l_2), \epsilon, \text{guard}, \text{update}, (l'_1, l'_2) \rangle \in T$  if and only if *at least* one of the following hold:
  - (a)  $\langle l_1, \epsilon, \text{guard}, \text{update}, l'_1 \rangle \in T_1$  and  $l_2 = l'_2$ .
  - (b)  $\langle l_2, \epsilon, \text{guard}, \text{update}, l'_2 \rangle \in T_2$ , and  $l_1 = l'_1$ .
- $\mathcal{F}_s = \{F^1, \dots, F^N\}$  such that for every  $F^i \in \mathcal{F}_s$ , either:
  - (a) there exists  $F_1^j \in \mathcal{F}_{s1}$  such that for every transition  $t \in T$ ,  $t \in F^i$  if and only if there exists a transition of the form  $\langle l_1, c, \text{guard}_1, \text{update}_1, l'_1 \rangle \in F_1^j$  and  $t$  is of the form  $\langle (l_1, l_2), c, \text{guard}_1 \wedge \text{guard}_2, \text{update}_1; \text{update}_2, (l'_1, l'_2) \rangle$  or of the form  $\langle (l_1, l_2), c, \text{guard}_1, \text{update}_1, (l'_1, l'_2) \rangle$ .
  - (b) there exists  $F_2^j \in \mathcal{F}_{s2}$  such that for every transition  $t \in T$ ,  $t \in F^i$  if and only if there exists a transition of the form  $\langle l_2, c, \text{guard}_2, \text{update}_2, l'_2 \rangle \in F_2^j$  and  $t$  is of the form  $\langle (l_1, l_2), c, \text{guard}_1 \wedge \text{guard}_2, \text{update}_1; \text{update}_2, (l'_1, l'_2) \rangle$ , or of the form  $\langle (l_1, l_2), c, \text{guard}_2, \text{update}_2, (l_1, l'_2) \rangle$ .
- $\mathcal{F}_w$  is defined in the same way as  $\mathcal{F}_s$

Note that the composition operator “ $\mid$ ” is commutative and associative.

**Specifications.** An ESM-S can be equipped with error locations  $L_e \subseteq L$ , accepting locations  $L_a \subseteq L$ , or both.<sup>1</sup> The composition of two ESM-S  $A_1, A_2$  “inherits” the error and accepting locations of its components. A product location  $(l_1, l_2)$  is an error (accepting) location if either  $l_1$  or  $l_2$  are error (accepting) locations.

<sup>1</sup>The error and accepting locations of an ESM-S are designated in the figures using double circles. Traditionally, *safety and liveness monitors* have been used to characterize erroneous finite and infinite executions. In this spirit, if an ESM-S is used solely for labeling product locations as error or accepting, we will call it a monitor, but still refer to the safety and liveness of the product ESM-S.

An ESM-S  $A$  is called *safe* under  $R$  if for all reachable states  $(l, \sigma)$ ,  $l$  is not an error location. An infinite execution of  $A$  under  $R$ ,  $(l_0, \sigma_0) \rightarrow (l_1, \sigma_1) \rightarrow \dots$ , is called *accepting* if for infinitely many indices  $j$ ,  $l_j \in L_a$ .  $A$  is called *live* under  $R$  if it has no infinite fair accepting executions.

### 3.2 Symmetry

It is often required that the processes of an ESM-S completion problem have some structurally similar behavior, as we saw in Section 2 in the case of Peterson’s algorithm. To describe such requirements, we use *symmetric types*, which are similar to *scalarsets* used in the Mur $\phi$  model checker [21].

A symmetric type  $T$  is characterized by: (a) its name, and (b) its cardinality  $|T|$ , which is a finite number. Given a collection of processes parameterized by a symmetric type  $T$ , such as  $P_0$  and  $P_1$  of Peterson’s algorithm, the idea is that the system is invariant under permutations (i.e. renaming) of the parameter values.

Let  $\text{perm}(T)$  be the set of all permutations  $\pi_T : T \rightarrow T$  over the symmetric type  $T$ . For ease of notation, we define  $\pi_T(v) = v$ , for values  $v$  whose type is *not*  $T$ . Given the collection of all symmetric types  $\mathcal{T} = \{T_1, T_2, \dots, T_n\}$  of the system, we can then describe permutations over  $\mathcal{T}$  as the composition of permutations over the individual types,  $\pi_{T_1} \circ \pi_{T_2} \circ \dots \circ \pi_{T_n}$ . Let  $\text{perm}(\mathcal{T})$  be the set of such “system-wide” permutations over  $\mathcal{T}$ .

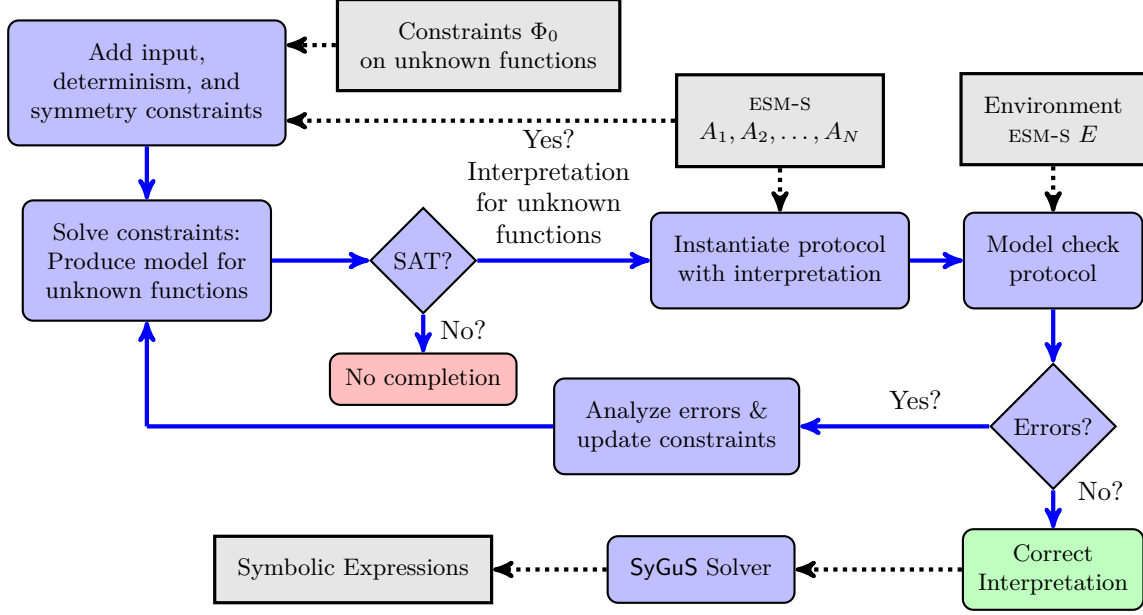
ESM sketches and input and output channels may thus be parameterized by symmetric values. The state variables and array variable indices<sup>2</sup> of an ESM-S may also be of symmetric type. Given the symmetric types  $\mathcal{T}$  and an interpretation  $R$  of the unknown functions in an ESM-S  $A$ , we say that  $A$  is *symmetric* with respect to  $\mathcal{T}$  if every execution  $(l_0, \sigma_0) \rightarrow (l_1, \sigma_1) \rightarrow \dots \rightarrow (l_n, \sigma_n) \rightarrow \dots$  of  $A$  under  $R$  also implies the existence of the permuted execution  $(\pi(l_0), \pi(\sigma_0)) \rightarrow (\pi(l_1), \pi(\sigma_1)) \rightarrow \dots \rightarrow (\pi(l_n), \pi(\sigma_n)) \rightarrow \dots$  of  $A$ , where the channel identifiers along transitions are also suitably permuted, for every permutation  $\pi \in \text{perm}(\mathcal{T})$ .

We therefore require that any interpretation  $R$  considered be such that the completed ESM-S  $A$  is symmetric with respect to  $\mathcal{T}$  under  $R$ . For every unknown function  $f$  in  $A$ , requiring that  $\forall d \in \text{dom}(f), \pi(f(d)) = f(\pi(d))$ , for each permutation  $\pi \in \text{perm}(\mathcal{T})$ , ensures that the behavior of  $f$  is symmetric. To see why, let us consider the example of the function  $f_{\text{turn}}$ , which could be used to update the variable *turn* in Peterson’s algorithm, shown in Figure 1(a). Suppose that the variable *turn* was to be updated as  $\text{turn} := f_{\text{turn}}(\text{Pm}, \text{Po}, \text{flag}, \text{turn})$ . Since there are only two permutations possible in this system ( $\text{Pm} = P_0, \text{Po} = P_1$ , and  $\text{Pm} = P_1, \text{Po} = P_0$ ), we would require, for example,  $f_{\text{turn}}(P_1, P_0, P_0, \langle \text{true}, \text{false} \rangle) = P_0 \leftrightarrow f_{\text{turn}}(P_0, P_1, P_1, \langle \text{false}, \text{true} \rangle) = P_1$ . Observe, that the two sides of the bi-directional implication can be obtained from each other by the permutation  $\pi \equiv \langle P_0 \mapsto P_1, P_1 \mapsto P_0 \rangle$ . In general we would add such constraints for each value in the range of the function, and for each permutation  $\pi \in \text{perm}(\mathcal{T})$ . Section 4, provides some additional examples to illustrate this. Note that while we have restricted the discussion here to only *full symmetry*, other notions of symmetry such as *ring symmetry* and *virtual symmetry* can also be accommodated into our formalization.

### 3.3 Completion Problem

In many cases, the designer has some prior knowledge about the unknown functions used in an ESM-S. For example, the designer may know that the variable *turn* is read-only during the  $(L_3, L_4)$  transition of Peterson’s algorithm. The designer may also know that the unknown guard of a transition is independent of some state variable. Many instances of such “prior knowledge” can already be expressed using the formalism just described: the update expression of *turn* in the unknown transition can be set to the identity function (in the first case), and the designer can omit the irrelevant variable from the signature of the update function (in the second case). We also allow the designer to specify additional constraints on the unknown functions: she may know, as in the case of Peterson’s algorithm for example, that  $g_{\text{crit}}(\text{Pm}, \text{Po}, \text{flag}, \text{turn}) \vee g_{\text{wait}}(\text{Pm}, \text{Po}, \text{flag}, \text{turn})$ ,

<sup>2</sup>To apply a permutation  $\pi$  to an array value, we first apply  $\pi$  to each element of the array, and then permute the indices of the array itself.



**Figure 2:** Completion Algorithm.

for every valuation of the function arguments  $\mathbf{Pm}$ ,  $\mathbf{Po}$ ,  $flag$ , and  $turn$ . This additional knowledge, which is helpful to guide the synthesizer, is encoded in the initial constraints  $\Phi_0$  imposed on candidate interpretations of  $U$ . Note that these constraints might refer to multiple unknown functions from the same or different ESM-S.

Formally, we can now state the completion problem as: Given a set of ESM-S  $A_1, \dots, A_N$  with sets of unknown functions  $U_1, \dots, U_N$ , an environment ESM-S  $E$  with an empty set of unknown functions, and a set of constraints  $\Phi_0$  on the unknown functions  $U = U_1 \cup \dots \cup U_N$ , find an interpretation  $R$  of  $U$ , such that (a)  $A_1, \dots, A_N$  are deterministic under  $R$ , (b) the completed system  $\Pi = A_1 \mid \dots \mid A_N \mid E$  is symmetric with respect to  $\mathcal{T}$  under  $R$ , where  $\mathcal{T}$  is the set of symmetric types in the system, (c)  $R$  satisfies the constraints in  $\Phi_0$ , and (d) the product  $\Pi$  under  $R$  is deadlock-free, safe, and live.

## 4 Solving the Completion Problem

The synthesis algorithm is outlined in Figure 2. We maintain a set of constraints  $\Phi$  on possible completions, and repeatedly query an SMT solver — Z3 [8] in our implementation — for candidate interpretations for the unknown functions satisfying all constraints in  $\Phi$ . If the protocol instantiated with the candidate interpretations is certified correct by the model checker, *i.e.*, the instantiated protocol satisfies all the safety and liveness requirements set forth by the programmer, then we are done. Otherwise, counter-example executions returned by the model checker are analyzed, and  $\Phi$  is strengthened with further constraints which eliminate all interpretations which result in similar erroneous executions from consideration in subsequent iterations of the algorithm. If a symbolic expression is required, we can submit the correct interpretation to a SyGuS solver [1]. A SyGuS solver takes a set of constraints  $\mathcal{C}$  on an unknown functions  $f$  together with the search space for the body of  $f$  — expressed as a grammar — and finds an expression in the grammar for  $f$ , such that it satisfies the constraints  $\mathcal{C}$ . In this section, we first describe the initial determinism and symmetry constraints expected of all completions. Next, we briefly describe the model checker used in our implementation, and then describe how to analyze counterexamples returned by the model checker. Finally, we describe additional heuristics to bias the SMT solver towards intuitively simpler completions.



## 4.1 Initial Constraints

**Determinism Constraints.** Recall that an ESM-S is deterministic under an interpretation  $R$  if and only if for every state  $(l, \sigma)$  if there are multiple transitions enabled at  $(l, \sigma)$ , then they must be input transitions on distinct input channels. We constrain the interpretations chosen at every step such that all ESM sketches in the protocol are deterministic. Consider the ESM-S for Peterson’s algorithm shown in Figure 1b. We have two transitions from the location  $L_3$ , with guards  $g_{\text{crit}}(\mathbf{Pm}, \mathbf{Po}, \text{flag}, \text{turn})$  and  $g_{\text{wait}}(\mathbf{Pm}, \mathbf{Po}, \text{flag}, \text{turn})$ . We ensure that these expressions never evaluate to true simultaneously with the constraint  $\neg \exists v_1 v_2 v_3 v_4 (g_{\text{crit}}(v_1, v_2, v_3, v_4) \wedge g_{\text{wait}}(v_1, v_2, v_3, v_4))$ . Although this is a quantified expression, which can be difficult for SMT solvers to solve, note that we only support finite types, whose domains are often quite small. So our tool unrolls the quantifiers and presents only quantifier-free formulas to the SMT solver.

**Symmetry Constraints.** Suppose that the interpretation chosen for the guard  $g_{\text{crit}}$  shown in Figure 1b, was such that  $g_{\text{crit}}(\mathbf{P0}, \mathbf{P1}, \langle \perp, \top \rangle, \mathbf{P0}) = \text{true}$ . Then for the ESM sketch to be symmetric under this interpretation, we require that  $g_{\text{crit}}(\mathbf{P1}, \mathbf{P0}, \langle \top, \perp \rangle, \mathbf{P1}) = \text{true}$  as well, because the latter expression is obtained by applying the permutation  $\{\mathbf{P0} \mapsto \mathbf{P1}, \mathbf{P1} \mapsto \mathbf{P0}\}$  on the former expression. Note that the elements of the *flag* array in the preceding example were flipped, because *flag* is an array indexed by the symmetric type *processid*. In general, given a function  $f \in U_i$ , we enforce the constraint  $\forall \pi \in \text{perm}(\mathcal{T}) \forall d \in \text{dom}(f) (f(\pi(d)) \equiv \pi(f(d)))$ , where  $\mathcal{T}$  is the set of symmetric types that appear in  $A_i$ . As in the case of determinism constraints, we unroll the quantifiers here as well.

## 4.2 Model Checker

To effectively and repeatedly generate constraints to drive the synthesis loop, a model checker needs to: (a) support checking liveness properties, with algorithmic support for fine grained notions of strong and weak fairness, (b) dynamically prioritize certain paths over others (*cf.* Section 4.4), and (c) exploit symmetries inherent in the model. The fine grained notions of fairness over sets of transitions, rather than bulk process fairness are crucial. For instance, in the case of unordered channel processes, we often require that no message be delayed indefinitely, which cannot be captured by enforcing fairness at the level of the entire process. The ability to prioritize certain paths over others is also crucial so that candidate interpretations are exercised to the extent possible in one model checking run (*cf.* Section 4.4). Finally, support for symmetry-based state space reductions, while not absolutely crucial, can greatly speed up each model checking run.

Surprisingly, we found that none of the well-supported model checkers met all of our requirements. SPIN [15] only supports weak process fairness at an algorithmic level and does not employ symmetry-based reductions. Our efforts to encode the necessary fine grained strong fairness requirements as LTL formulas in SPIN resulted in the Büchi monitor construction step either blowing up or generating extremely large monitor processes. Support for symmetry-based reductions is present in Mur $\varphi$  [10, 21], but it lacks support for liveness checking.<sup>3</sup> SMC [26] is a model checker with support for symmetry reduction and strong and weak process fairness. Unfortunately, it is no longer maintained, and has very rudimentary counterexample generation capabilities. Finally, NuSMV [7] does not support symmetry reductions, but supports strong and weak process level fairness. But, due to bugs in the implementation of counterexample generation, we were unable to obtain counterexamples in some cases.

We therefore implemented a model checker based on the ideas used in Mur $\varphi$  [10] for symmetry reduction, and an adaptation of the techniques presented in [11] for checking liveness properties under fine grained fairness assumptions. We plan on releasing the model checker as standalone open-source tool in the near future. The model checking algorithm consists of the following steps:

<sup>3</sup>There exists an unmaintained version of Mur $\varphi$  which does support checking of some restricted forms of LTL properties, but it only supports weak fairness.

1. Construct the symmetry reduced state graph in the form of the *Annotated Quotient Structure* described in earlier literature [11]. In our setting, we treat the liveness monitors the same way as other processes in the system. So this graph is really the symmetry reduced *product structure* described in earlier literature [11]. The task is to find *fair, accepting* cycles in this symmetry reduced product graph.
2. To accomplish this search for fair cycles in a sound and complete way, we implicitly search over the *threaded* graph described in earlier literature [11]. This threaded graph annotates each state with an additional component which keeps track of the permutations that have been applied along a path, which will then be used to “adjust” the fairness requirements that have already been satisfied. This is necessary, because the product state space is *compressed*, as a result, a path in the product state space could correspond to more than one uncompressed paths. We refer the reader to published literature [11] for more details on the construction of the *threaded* graph structure.
3. Once the *threaded* graph has been constructed, we then simply compute the accepting strongly connected components (SCCs) in this threaded graph using Tarjan’s algorithm for computing strongly connected components in a directed graph [25]. An *accepting* SCC is simply an SCC which contains at least one accepting state.
4. In each accepting SCC  $C$ , where all the weak fairness requirements are satisfied, and for each strong fairness requirement  $F_s$  which is not satisfied in  $C$ , *i.e.*, some transition in  $F_s$  is *enabled* in  $C$ , but no transition in  $F_s$  is ever *taken* in  $C$ , we delete from the threaded graph, all the states where some transition in  $F_s$  is enabled.
5. Steps (3) and (4) are repeated until either (a) an accepting SCC  $C$  is found which satisfies all the strong and weak fairness requirements, in which case we can construct a counterexample from  $C$ , or (b) no more accepting SCCs remain, in which case the protocol satisfies all the liveness requirements and no *fair, accepting* execution exists.

### 4.3 Analysis of Counterexamples

We now describe our algorithms for analyzing counterexamples by way of examples first and then provide a formal description of the algorithms.

**Analyzing deadlocks, an example.** In Figure 1b, consider the candidate interpretation where both  $g_{\text{crit}}$ ,  $g_{\text{wait}}$  are set to be universally false. Two deadlock states are then reachable:  $S_1 = ((L_3, L_3), \{flag \mapsto \langle \top, \top \rangle, turn \mapsto P1\})$  and  $S_2 = ((L_3, L_3), \{flag \mapsto \langle \top, \top \rangle, turn \mapsto P0\})$ . We strengthen  $\Phi$  by asserting that these deadlocks do not occur in future interpretations: either  $S_1$  is unreachable, or the system can make a transition from  $S_1$  (and similarly for  $S_2$ ). In this example, the reachability of both deadlock states is not dependent on the interpretation, *i.e.*, the execution that leads to the states does not exercise any unknown function, hence, we need to make sure that the states are not deadlocks. The possible transitions out of location  $(L_3, L_3)$  are the transitions from  $L_3$  to  $L_3$  (waiting transition) and from  $L_3$  to  $L_4$  (critical transition) for each of the two processes. In each deadlock state, at least one of the four guards has to be true:  $g_{\text{wait}}(P0, P1, \langle \top, \top \rangle, P1) \vee g_{\text{crit}}(P0, P1, \langle \top, \top \rangle, P1) \vee g_{\text{wait}}(P1, P0, \langle \top, \top \rangle, P1) \vee g_{\text{crit}}(P1, P0, \langle \top, \top \rangle, P1)$  for  $S_1$ , and  $g_{\text{wait}}(P0, P1, \langle \top, \top \rangle, P0) \vee g_{\text{crit}}(P0, P1, \langle \top, \top \rangle, P0) \vee g_{\text{wait}}(P1, P0, \langle \top, \top \rangle, P0) \vee g_{\text{crit}}(P1, P0, \langle \top, \top \rangle, P0)$  for  $S_2$ . The two disjunctions are added to the set of constraints, since any candidate interpretation has to satisfy them in order for the resulting product to be deadlock-free.

**Analyzing safety violations, an example.** Consider now an erroneous interpretation where the critical transition guards are true for both processes when  $turn$  is  $P0$ , that is:  $g_{\text{crit}}(P0, P1, \langle \top, \top \rangle, P0)$  and  $g_{\text{crit}}(P1, P0, \langle \top, \top \rangle, P0)$  are set to true. Under this interpretation the product can reach the error location  $(L_4, L_4)$ . We perform a weakest precondition analysis on the corresponding execution to obtain a necessary condition under which the safety violation is possible. In this case, the execution crosses both critical transitions and the generated constraint is  $\neg g_{\text{crit}}(P0, P1, \langle \top, \top \rangle, P0) \vee \neg g_{\text{crit}}(P1, P0, \langle \top, \top \rangle, P0)$ . Note that the

conditions obtained from this analysis are necessary; the product under any interpretation that does not satisfy them will exhibit the same safety violation.

**Analyzing liveness violations, an example.** An interpretation that satisfies the constraints gathered above is one that, when  $turn$  is  $\mathbf{P0}$ , enables both waiting transitions and disables the critical ones. Intuitively, under this interpretation, the two processes will not make progress if  $turn$  is  $\mathbf{P0}$  when they reach  $L_3$ . The executions in which the processes are at  $L_3$  and either  $P_0$  or  $P_1$  continuously take the waiting transition is an accepting one. As with safety violations, we eliminate liveness violations by adding constraints generated through weakest precondition analysis of the accepting executions. In this case, this results in two constraints:  $\neg g_{\text{wait}}(\mathbf{P0}, \mathbf{P1}, \langle \top, \top \rangle, \mathbf{P0})$  and  $\neg g_{\text{wait}}(\mathbf{P1}, \mathbf{P0}, \langle \top, \top \rangle, \mathbf{P0})$ . However, in the presence of fairness assumptions, these constraints are too strong. This is because removing an execution that causes a fair liveness violation is not the only way to resolve it: another way is to make it unfair. Given the weak fairness assumption on the transitions on the critical <sub>$P_i$</sub>  channels, the correct constraint generated for the liveness violation of Process  $P_0$  is:  $\neg g_{\text{wait}}(\mathbf{P0}, \mathbf{P1}, \langle \top, \top \rangle, \mathbf{P0}) \vee g_{\text{crit}}(\mathbf{P0}, \mathbf{P1}, \langle \top, \top \rangle, \mathbf{P0}) \vee g_{\text{crit}}(\mathbf{P1}, \mathbf{P0}, \text{true}, \text{true}, \mathbf{P0})$ , where the last two disjuncts render the accepting execution unfair.

We now describe in detail how we perform the analysis of counterexamples returned by the model checker. Our implementation first composes the ESM sketches to form a *product* ESM-S II. It then compiles down this product ESM-S II into guarded commands, where the guards and updates are as defined in Section 3.1. The guards and updates of the guarded commands are also transformed by the compiler to use **select**, **store**, **project** and **record update** functions<sup>4</sup> for reads and updates of arrays and records respectively. Furthermore, repeated assignments to the same variable in a guarded command are coalesced into a single assignment. In effect, each variable (be it of a scalar type, an array type or a record type) have *only one* assignment to it in the list of updates associated with each guarded command. These transformations on the guarded commands make it easier to compute the weakest preconditions of predicates with respect to the guarded commands, as we shall now explain.

Let the set of guarded commands be  $G$ , given a guarded command  $\text{cmd} \in G$ , we define  $\text{guard}(\text{cmd})$  to be the guard of  $\text{cmd}$  and  $\text{update}(\text{cmd})$  to be the list of coalesced updates of  $\text{cmd}$ . The weakest precondition of a predicate  $\varphi$  with respect to an assignment statement  $\text{stmt} \triangleq l := e$  is defined as  $\text{wp}(\text{stmt}, \varphi) \equiv \varphi[l \leftarrow e]$ , where  $\varphi[l \leftarrow e]$  is the expression obtained by replacing all instances of the sub-expression  $l$  in  $\varphi$  with the expression  $e$ . We extend the definition of the weakest precondition of a predicate  $\varphi$  with respect to a sequence of statements in the natural way. The weakest precondition of a predicate  $\varphi$  with respect to a guarded command  $\text{cmd}$  is defined as  $\text{wpcmd}(\text{cmd}, \varphi) \equiv \text{guard}(\text{cmd}) \rightarrow \text{wp}(\text{update}(\text{cmd}), \varphi)$ .

**Analyzing Safety and Deadlock Counterexamples.** Given an error trace (*i.e.*, a witness for a safety violation or a deadlock) which consists of an initial state valuation  $\sigma_0$ , and a sequence of guarded commands from  $G$ , say,  $\text{cmd}_1, \text{cmd}_2, \dots, \text{cmd}_n$ . We define  $\text{pre}_0(\varphi) \equiv \varphi$ , and recursively define  $\text{pre}_i(\varphi) \equiv \text{wpcmd}(\text{cmd}_{n-i-1}, \text{pre}_{i-1}(\varphi))$ . Then, if the trace is a witness for a safety violation, we add the constraint  $\text{pre}_n(\text{false})[v \leftarrow \sigma_0(v)]$ , for every  $v \in V_1 \cup V_2 \cup \dots \cup V_n$ , to our set of constraints  $\Phi$ , which essentially ensures that the particular execution is no longer possible under all interpretations chosen for the set of unknown functions  $U$  in the future. On the other hand if the trace is a witness for a deadlock, we add  $\text{pre}_n(\bigvee_{\text{cmd} \in G} \text{guard}(\text{cmd})) [v \leftarrow \sigma_0(v)]$ , for every  $v \in V_1 \cup V_2 \cup \dots \cup V_n$ , to  $\Phi$ . This constraint ensures that if this particular execution is ever permitted under an interpretation for the unknown functions  $U$  chosen in the future, then some guarded command is enabled at the end of the execution, under that interpretation, therefore no longer rendering the final state of the execution a deadlock.

**Analyzing Liveness Counterexamples.** We assume that infinite accepting executions are given as a pair of a finite stem execution of size  $n$  and a finite cycle execution of size  $m$ . First, we describe the case where no fairness assumptions exist in the system. The constraint computed from an accepting execution asserts either that the sequence of transitions should not be enabled or that the state of the system at the beginning of the

<sup>4</sup>These are functions defined in the theory of arrays and records by the SMTLIB2 standard. For details, see <http://smt-lib.org/>

cycle should be not be the same as the state at the end. If the set of variables of  $\Pi$  is  $\{s_1, \dots, s_N\}$  we introduce symbolic constants  $s'_1, \dots, s'_N$  and set  $\phi \equiv s_1 \neq s'_1 \vee s_2 \neq s'_2 \vee \dots \vee s_N \neq s'_N$ . We first compute  $\phi' = \text{pre}_m(\phi)$  on the cycle execution and then substitute  $s'_1, \dots, s'_N$  for  $s_1, \dots, s_N$  in  $\phi'$ :  $\phi'' = \phi'[s'_1 \leftarrow s_1, \dots, s'_N \leftarrow s_N]$ . We then get the final constraint by computing  $\text{pre}_n(\phi'')$  on the stem execution.

We now describe the case where strong fairness assumptions are present. The treatment of weak fairness assumptions is similar. Let  $\mathcal{F}$  be the set of strong fairness assumptions and  $G$  be the union of all fairness sets  $F \in \mathcal{F}$  such that every guarded command in  $F$  is disabled in the cycle. We adapt the computation of  $\text{pre}_i$  in the cycle execution as follows:  $\text{pre}'_i(\varphi) \equiv \text{wpcmd}(\text{cmd}_{n-i-1}, \text{pre}_{i-1}(\varphi) \vee \bigvee_{\text{cmd} \in F} \text{guard}(\text{cmd}))$ . Enabling a command  $\text{cmd}$  in  $G$  at a step in the cycle execution has the effect of making the accepting cycle unfair: since  $\text{cmd}$  is never executed in the cycle, enforcing  $\text{guard}(\text{cmd})$  makes  $\text{cmd}$  infinite often enabled but never taken.

## 4.4 Optimizations and Heuristics.

We describe a few key optimizations and heuristics that improve the scalability and predictability of our technique.

**Not all counterexamples are created equal.** The constraint we get from a single counter-example trace is weaker when it exercises a large number of unknown functions. Consider, for example, a candidate interpretation for the incomplete Peterson's algorithm which, when  $\text{turn} = \mathbf{P0}$ , sets both waiting transition guards  $g_{\text{wait}}$  to **true**, and both critical transition guards  $g_{\text{crit}}$  to **false**. We have already seen that the product is not live under this interpretation. From the infinite execution leading up-to the location  $(L_3, L_3)$ , and after which  $P_0$  loops in  $L_3$ , we obtain the constraint<sup>5</sup>  $\neg g_{\text{wait}}(\mathbf{P0}, \mathbf{P1}, \langle \top, \top \rangle, \mathbf{P0})$ . On the other hand, if we had considered the longer self-loop at  $(L_3, L_3)$ , where  $P_0$  and  $P_1$  alternate in making waiting transitions, we would have obtained the weaker constraint  $\neg g_{\text{wait}}(\mathbf{P0}, \mathbf{P1}, \langle \top, \top \rangle, \mathbf{P0}) \vee \neg g_{\text{wait}}(\mathbf{P1}, \mathbf{P0}, \langle \top, \top \rangle, \mathbf{P0})$ . In general, erroneous traces which exercise fewer unknown functions have the potential to prune away a larger fraction of the search space and are therefore preferable over traces exercising a larger number of unknown functions.

In each iteration, the model checker discovers several erroneous states. In the event that the candidate interpretation chosen is blatantly incorrect, it is infeasible to analyze paths to all error states. A naïve solution would be to analyze paths to the first  $n$  errors states discovered (where  $n$  is configurable). But depending on the strategy used to explore the state space, a large fraction these errors could be similar<sup>6</sup>, and would only provide us with rather weak constraints. On the other hand, exercising as many unknown functions as possible, along different paths, has the potential to provide stronger constraints on future interpretations. In summary, we bias the model checker to *cover* as many unknown functions as possible, such that each path exercises as few unknown functions as possible.

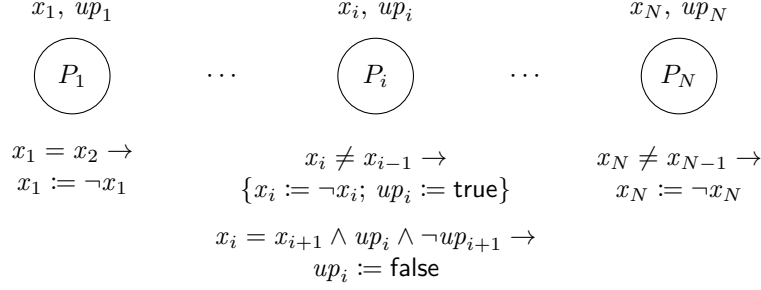
**Heuristics/Prioritizations to guide the SMT solver.** As mentioned earlier, we use an SMT solver to obtain interpretations for unknown functions, given a set of constraints. When this set is small, as is the case at the beginning of the algorithm, there exist many satisfying interpretations. At this point the interpretation chosen by the SMT solver can either lead the rest of the search down a “good” path, or lead it down a futile path. Therefore the run time of the synthesis algorithm can depend heavily on the interpretations returned by the SMT solver, which we consider a non-deterministic black box in our approach.

To reduce the influence of non-determinism of the SMT solver on the run time of our algorithm, we bias the solver towards specific forms of interpretations by asserting additional constraints. These constraints associate a *cost* with interpretations and require an interpretation with a given bound on the cost, which is relaxed whenever the SMT solver fails to find a solution.

We briefly describe the most important of the heuristics/prioritization techniques: (1) We minimize the number of points in the domain of an unknown guard function at which it evaluates to true. This results in minimally permissive guards. (2) Based on the observation that most variables are unchanged in a given

<sup>5</sup>Ignoring fairness assumptions.

<sup>6</sup>We observed this phenomenon in our initial experiments.



**Figure 3:** Self-stabilizing system processes.

transition, we prioritize interpretations where update functions leave the value of the variable unchanged, as far as possible. (3) In the event that the value of the variable cannot be left unchanged, we try to minimize the number of arguments on which the value of the unknown function depends, in an attempt to bias the SMT solver towards *intuitively* simple interpretations.

## 5 Experiments

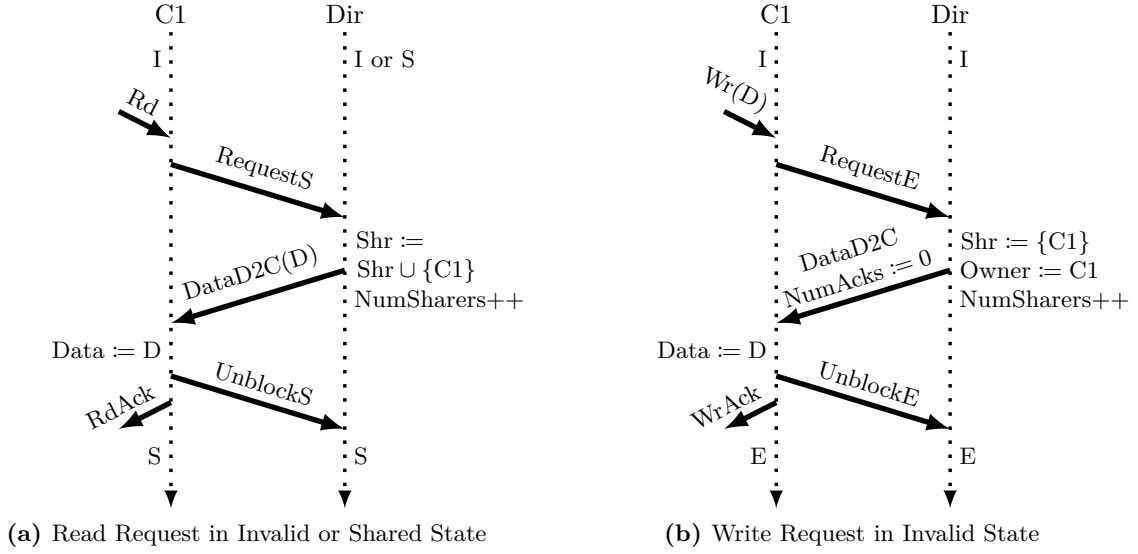
### 5.1 Peterson’s Mutual Exclusion Protocol

In addition to the missing guards  $g_{\text{crit}}$  and  $g_{\text{wait}}$ , we also replace the update expressions of  $\text{flag}[\text{Pm}]$  in the  $(L_1, L_2)$  and  $(L_4, L_1)$  transitions with unknown functions that depend on all state variables. In the initial constraints we require that  $g_{\text{crit}}(\text{Pm}, \text{Po}, \text{flag}, \text{turn}) \vee g_{\text{wait}}(\text{Pm}, \text{Po}, \text{flag}, \text{turn})$ . The synthesis algorithm returns with an interpretation in less than a second. Upon submitting the interpretation to a SyGuS solver, the synthesized expressions match the ones shown in Figure 1b.

### 5.2 Self-stabilizing Systems

Our next case study is the synthesis of self-stabilizing systems [9]. A distributed system is self-stabilizing if, starting from an arbitrary initial state, in each execution, the system eventually reaches a global *legitimate* state, and only legitimate states are ever visited after. We also require that every legitimate state be reachable from every other legitimate state. Consider  $N$  processes connected in a line. Each process maintains two Boolean state variables  $x$  and  $up$ . The processes are described using guarded commands of the form, “**if** *guard* **then** *update*”. Whether a command is enabled is a function of the variable values  $x$  and  $up$  of the process itself, and those of its neighbors. We attempted to synthesize the guards and updates for the middle two processes of a four process system  $P_1, P_2, P_3, P_4$ . Specifically, the ESM-s for  $P_2$  and  $P_3$  have two transitions, each with an unknown function as a guard and two unknown functions for updating its state variables. The guard is a function of  $x_{i-1}, x_i, x_{i+1}, up_{i-1}, up_i, up_{i+1}$ , and the updates of  $x_i$  and  $up_i$  are functions of  $x_i$  and  $up_i$ . We followed the definition in [14] and defined a state as being legitimate if exactly one guarded command is enabled globally. We also constrain the completions of  $P_2$  and  $P_3$  to be identical. The complete self-stabilizing system is shown in Figure 3. In our experiment we synthesized the guards and updates of processes  $P_2$  and  $P_3$  in a four process system, i.e.,  $N = 4$ .

Due to the large number of unknown functions needed to be synthesized in this experiment and, in particular, because there were a lot of input domain points at which the guards had to be true, the heuristic that prefers minimally permissive guards, described in Section 4, was not effective. However, the heuristic that prioritizes interpretations in which the guards depend on fewer arguments of their domain was effective.



**Figure 4:** Simple Cases for Read and Write Requests

For state variable updates, we applied the heuristic that prioritizes functions that leave the state unchanged or set it to a constant. After passing the synthesized interpretation through a SyGuS solver, the expressions we got were exactly the same as the ones found in [9], and presented in Figure 3.

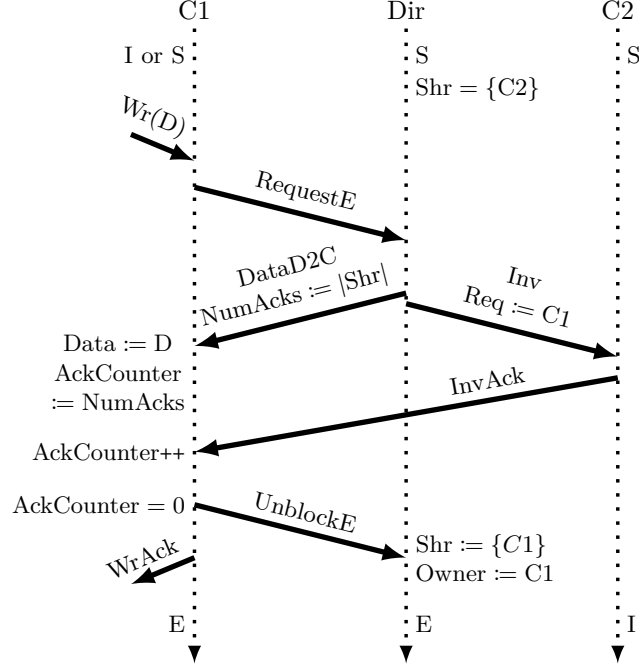
### 5.3 Cache Coherence Protocol

A cache coherence protocol ensures that the copies of shared data in the private caches of a multiprocessor system are kept up-to-date with the most recent version. We describe the working of a variant of the German cache coherence protocol, which is often used as a case study in model checking research [6, 29]. The protocol consists of a *Directory* process,  $n$  symmetric *Cache* processes and  $n$  symmetric *Environment* processes, one for each cache process. Each cache may be in the E, S or I state, indicating read-write, read, and no permissions on the data respectively. All communication between the caches and the directory is non-blocking, and occurs over buffered, unordered communication channels.

Figure 4(a) shows the actions performed by the various processes when a cache receives a *read* command from its environment. It sends a *RequestS* message to the directory. In this particular scenario, the directory has recorded that all other caches are either in the I or S state, and proceeds to send the most up-to-date copy of the data in a *DataD2C* message. The cache then updates its local copy of the data, notifies its environment that the request has been satisfied and transitions to the S state.

Figure 4(b) shows what happens when a cache receives a *write* command from its environment along with the new data value  $D$  to write. In this particular case, the directory knows that all other caches are in the I state and thus proceeds to acknowledge the *RequestE* message from the cache with a *DataD2C* message which also contains the number of acknowledgments the cache needs to wait for before gaining write permissions on the data. In this case, since all other caches are in the I state, the number of acknowledgments to wait for is zero. The cache therefore, immediately updates its local copy of the data with the new value  $D$  and notifies its environment that the request has been satisfied and transitions to the E state.

On the other hand, Figure 5 depicts the scenario when a *write* command is received by a cache and some other cache is in the S state. In this case, the directory sends invalidations to all the caches in the S state, and sends a *DataD2C* message to the requesting cache with the *NumAcks* field set to the number of



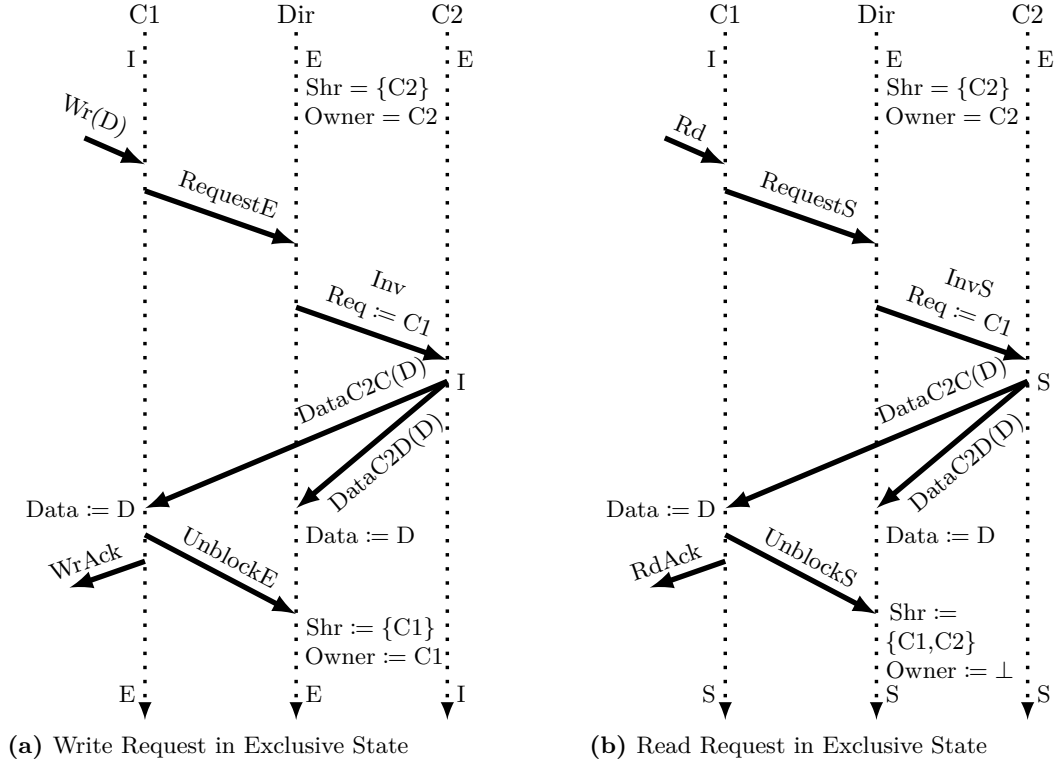
**Figure 5:** Write Request in Shared State

sharers, notifying the cache that it needs to wait for as many invalidate acknowledgments. The other caches directly communicate with the requesting cache by sending acknowledgment of the invalidation from the directory. Note that this is not part of the base German/MSI coherence protocol, where the directory collects acknowledgments instead. With the extension, the cache-to-cache communication reduces the amount of processing that needs to be done in the centralized directory.

Figure 6(a) describes the behavior of the protocol when a cache receives a *write* request and some other cache in the system is in the E state. The actions are similar to the case where some other cache is in the S state, except that the cache already in the E state directly sends its data to the requesting cache, as well as to the directory. And the requesting cache does not need to wait for any acknowledgments. Note that this is again an extension to the base German/MSI protocol, where the data is sent to only the directory, and the directory forwards the data back to the requesting cache. Again, this extension reduces the amount of processing that needs to be handled at the centralized directory.

The scenario when a cache receives a *read* command from the environment when some other cache in the system is in the E state is shown in Figure 6(b). Again, the directory sends an invalidation to the cache in the E state, which in turn responds by sending the most up-to-date copy of the data to the directory as well as the requesting cache. It then downgrades its permissions to the S state. Both the cache and the directory update their local copies of the state. The directory in addition adds the requesting cache to set of sharers.

Figures 7(a) and 7(b) describe the behavior of the protocol in the case where a cache wishes to relinquish its permissions. This is not a scenario that occurs in the base German/MSI protocol, but is necessary in a real-world coherence protocol, where a line of unused data may need to be evicted to make room for some other data. In the event that the cache is in the S state, it silently evicts the line, without notifying the directory. This can be done, only because the directory already has the most up-to-date copy of the data — recall that the S state only grants read permissions to the cache, hence it could not have modified the data. On the other hand if the cache is in the E state, then it needs to send the most up-to-date copy of the data to the directory. Therefore it sends a *WriteBack* message to the directory which contains the most up-to-date copy



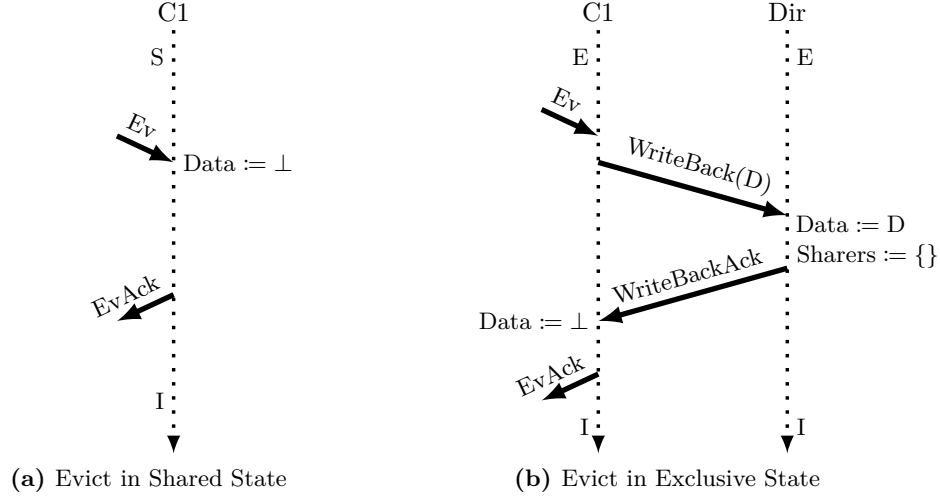
**Figure 6:** Requests in Exclusive State in the German/MSI Protocol

of the data. The directory then updates its local copy of the data with this copy and notes that all caches in the system are in the I state.

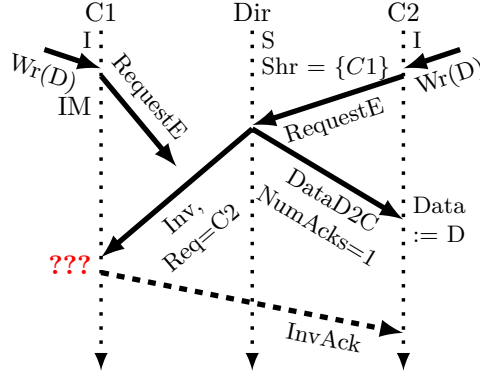
**Corner-cases in the German/MSI Protocol.** We consider a more complex variant of the German cache coherence protocol to evaluate the techniques we have presented so far, which we refer to as German/MSI. The main differences from the base German protocol are: (1) Direct communication between caches is possible in some cases, (2) A cache in the S state can silently relinquish its permissions, which can cause the directory to have out-of-date information about the caches which are in the S state. (3) A cache in the E state can coordinate with the directory to relinquish their permissions. We have discussed a list of scenarios typically used when describing this protocol. These scenarios however, do not describe the protocol's behavior in several cases induced by concurrency. There are five such cases that need to be considered in the case of the German/MSI protocol.

Figure 8, presents the first of these cases. Initially, cache C1 is in the I state, in contrast, the directory records that C1 is in state S and is a sharer, due to C1 having silently relinquished its read permissions at some point in the past. Now, both caches C1 and C2 receive *write* commands from their respective environments. Cache C2 sends a *RequestE* message to the directory, requesting exclusive write permissions. The directory, under the impression that C1 is in state S, sends an *Inv* message to it, informing it that C2 has requested exclusive access and C1 needs to acknowledge that it has relinquished permissions to C2. Concurrently, cache C1 sends a *RequestE* message to the directory requesting write permissions as well, which gets delayed. Subsequently, the cache C1 receives an invalidation when it is in the state IM, which cannot happen in the base German protocol. The correct behavior for the cache in this situation (shown by dashed arrows), is to send an *InvAck* message to the cache C2. The guard, the state variable updates, as well as





**Figure 7:** Evict requests in the German/MSI protocol

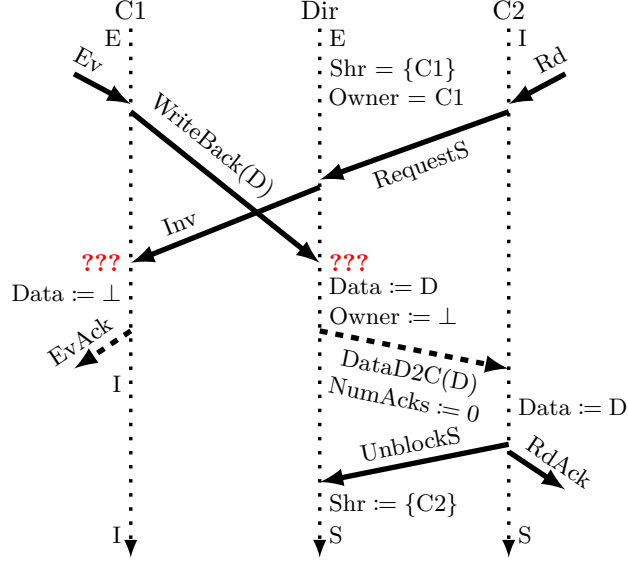


**Figure 8:** Racy Scenario

the location update is what we have left unspecified in the case of this particular scenario. As part of the evaluation, we successfully synthesized the behavior of the German/MSI protocol in five such corner-case scenarios arising from concurrency.

Two of the remaining four scenarios are similar to the one shown in Figure 8, with the only difference being that either the *RequestS* message is sent by C1 in response to a *Read* command from the environment, or that C1 begins in the S state, and sends a *RequestE* message in response to a *Write* command.

We now describe the last two corner-cases which arise from concurrency, in the German/MSI protocol. These are depicted in Figure 9. Essentially, the scenarios shown in Figures 7(b) and 6(b) interleave, to obtain the situation shown in Figure 9. The cache C1 having sent a *WriteBack* message to the directory is not expecting an *Inv* message. Similarly, the directory, having sent an *Inv* to cache C1 is not expecting a *WriteBack* message from it. The correct way for the processes to behave in this situation is shown by dashed arrows in Figure 9. The cache behaves as if the *Inv* message was a *WritebackAck* message and notifies its environment of completion. The directory updates its local copy of the data with the one from the *WriteBack* message, and then sends this data over to the cache C2, informing it that it need not wait for any acknowledgment. After this point, both the cache and directory behaviors know how to interact with each other as shown in Figure 4(a). For completeness, the way the scenario plays out is shown in Figure 9 as well.



**Figure 9:** An interleaving of scenarios which leads to unspecified behavior in the MSI/German protocol

## 5.4 Summary of Experimental Results

Table 1 summarizes our experimental findings. All experiments were performed on a Linux desktop, with an Intel Core i7 CPU running at 3.4 GHz., with 8 GB of memory. The columns show the name of the benchmark, the number of unknown functions that were synthesized ( $\#$  UF), the size of the search space for the unknown functions, the number of states in the complete protocol ( $\#$  States), “*symm. red.*” denotes symmetry reduced state space. The “ $\#$  Iters.” column shows the number of algorithm iterations, while the last two columns show the total amount of time spent in SMT solving and the end-to-end synthesis time.

The “German/MSI- $n$ ” rows correspond to the synthesizing the unknown behavior for the German/MSI protocol, with  $n$  out of the five unknown transitions left unspecified. In each case, we applied the heuristic to obtain minimally permissive guards and biased the search towards updates which leave the values of state variables unchanged as far as possible, except in the case of the Dijkstra benchmark, as mentioned in Section 5.2. Also, note that we ran each benchmark multiple times with different random seeds to the SMT solver, and report the worst of the run times in Table 1.

**Programmer Assistance.** In all cases, the programmer specified the kinds of messages to handle in the states where the behavior was unknown. For example, in the case of the German/MSI protocol, the programmer indicated that in the IM state on the cache, it needs to handle an invalidation from the directory (see Figure 8). In general, the programmer specified *what* needs to be handled, but not the *how*. This was crucial to getting our approach to scale.

**Overhead of Decision Procedures.** We observe from Table 1 that for the longer running benchmarks, the run time is dominated by SMT solving. In all of these cases, a very large fraction of the constraints asserted into the SMT solver are constraints to implement heuristics which are specifically aimed at guiding the SMT solver, and reducing the impact of non-deterministic choices made by the solver. Specialized decision procedures that handle these constraints at an algorithmic level [3] can greatly speed up the synthesis procedure.

**Synthesizing Symbolic Expressions.** The interpretations returned by the SMT solver are in the form of tables, which specify the output of the unknown function on specific inputs. We mentioned that if a symbolic expression is required we can pass this output to a SyGuS solver, which will then return a symbolic expression.

Benchmark	# UF	Search Space	# States	# Iters.	SMT Time	Total Time
Peterson	3	$2^{36}$	60	14	97ms	130ms
Dijkstra	6	$2^{192}$	$\sim 2000$	30	27s	64s
German/MSI-2	16	$\sim 2^{4700}$	$\sim 20000$ (symm. red.)	217	31s	298s
German/MSI-4	28	$\sim 2^{7614}$	$\sim 20000$ (symm. red.)	419	898s	1545s
German/MSI-5	34	$\sim 2^{9000}$	$\sim 20000$ (symm. red.)	525	2261s	3410s

**Table 1:** Experimental Results

We were able to synthesize compact expressions in all cases using the enumerative SyGuS solver [1]. Further, although the interpretations are only guaranteed to be correct for the finite instance of the protocol, the symbolic expressions generated by the SyGuS solver were *parametric*. We found that they were general enough to handle larger instances of protocol as well.

## 6 Conclusions

We have presented an algorithm to complete symmetric distributed protocols specified as ESM sketches, such that they satisfy the given safety and liveness properties. A prototype implementation, which included a custom model checker, successfully synthesized non-trivial portions of Peterson’s mutual exclusion protocol, Dijkstra’s self-stabilizing system, and the German/MSI cache coherence protocol. We show that programmer assistance in the form of *what* needs to be handled is crucial to the scalability of the approach. Scalability is currently limited by the constraints that require candidate interpretations to be intuitively simple. Note that these heuristics were necessary to reduce the dependence of the run time of our algorithm on the non-deterministic choices made by the SMT solver. As part of future work, we plan to investigate decision procedures that implement these heuristics at an algorithmic level, rather than using constraints which are treated the same as the correctness constraints.

## References

- [1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided Synthesis. In *FMCAD*, pages 1–17, 2013.
- [2] Rajeev Alur, Milo M. K. Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing Finite-State Protocols from Scenarios and Requirements. In Eran Yahav, editor, *10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, volume 8855 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2014. ISBN 978-3-319-13337-9. doi: 10.1007/978-3-319-13338-6.
- [3] Nikolaj Björner and Anh-Dung Phan.  $\nu Z$  - Maximal Satisfaction with Z3. In Temur Kutsia and Andrei Voronkov, editors, *SCSS 2014*, volume 30 of *EPiC Series*, pages 1–9. EasyChair, 2014.
- [4] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa’ar. Synthesis of Reactive(1) Designs. *J. Comput. Syst. Sci.*, 78(3), 2012.
- [5] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning Extended Finite State Machines. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, volume 8702 of *Lecture Notes in Computer Science*, pages 250–264. Springer International Publishing, 2014. ISBN 978-3-319-10430-0. doi: 10.1007/978-3-319-10431-7\_18.

- [6] Ching-Tsun Chou, PhanindraK. Mannava, and Seungjoon Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 382–398. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23738-9. doi: 10.1007/978-3-540-30494-4\_27.
- [7] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NUSMV 2: An Open-source Tool for Symbolic Model Checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43997-4. doi: 10.1007/3-540-45657-0\_29.
- [8] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78799-0.
- [9] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11): 643–644, November 1974. ISSN 0001-0782. doi: 10.1145/361179.361202.
- [10] David L. Dill. The Mur $\varphi$  Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV ’96, pages 390–393, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61474-5.
- [11] E. Allen Emerson and A. Prasad Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, 1997. doi: 10.1145/262004.262008.
- [12] B. Finkbeiner and S. Schewe. Uniform Distributed Synthesis. In *IEEE Symposium on Logic in Computer Science*, pages 321–330, 2005.
- [13] B. Finkbeiner and S. Schewe. Bounded synthesis. *Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.
- [14] Adria Gascón and Ashish Tiwari. Synthesis of a Simple Self-stabilizing System. In *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, pages 5–16, 2014. doi: 10.4204/EPTCS.157.5.
- [15] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. ISSN 0098-5589. doi: 10.1109/32.588521.
- [16] B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *Computer Aided Verification, 17th International Conference*, LNCS 3576, pages 226–238, 2005.
- [17] G. Katz and D. Peled. Model Checking-Based Genetic Programming with an Application to Mutual Exclusion. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference*, LNCS 4963, pages 141–156, 2008.
- [18] Gal Katz and Doron Peled. Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming. In *Haifa Verification Conference*, pages 117–132, 2009.
- [19] H. Lamouchi and J. Thistle. Effective Control Synthesis for DES Under Partial Observations. In *39th IEEE Conference on Decision and Control*, pages 22–28, 2000.
- [20] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.
- [21] C. Norris IP and David L. Dill. Better Verification through Symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996. ISSN 0925-9856. doi: 10.1007/BF00625968.
- [22] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, 1989.
- [23] A. Pnueli and R. Rosner. Distributed Reactive Systems Are Hard to Synthesize. In *31st Annual*

*Symposium on Foundations of Computer Science*, pages 746–757, 1990.

- [24] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *IEEE Transactions on Control Theory*, 77:81–98, 1989.
- [25] Robert E. Tarjan. Depth first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1972.
- [26] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: A Symmetry-based Model Checker for Verification of Safety and Liveness Properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000. doi: 10.1145/350887.350891.
- [27] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by Sketching for Bitstreaming Programs. In *Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation*, 2005.
- [28] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching Concurrent Data Structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08.
- [29] Murali Talupur and Mark R. Tuttle. Going with the Flow: Parameterized Verification Using Message Flows. In Alessandro Cimatti and Robert B. Jones, editors, *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–8. IEEE, 2008. ISBN 978-1-4244-2735-2. doi: 10.1109/FMCAD.2008.ECP.14.
- [30] S. Tripakis. Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters*, 90(1):21–28, April 2004.
- [31] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 287–296, 2013.